Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation

Yuanjie Huang, Liang Peng, Chengyong Wu, Yuriy Kashnikov, Jörn Rennecke, Grigori Fursin

Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation

Yuanjie Huang[1,2], Liang Peng[1,2], Chengyong Wu[1]
Yuriy Kashnikov[4], Jörn Rennecke[3], Grigori Fursin[3]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2] Graduate School of the Chinese Academy of Sciences, Beijing, China
[3] INRIA Saclay, Orsay, France (HiPEAC member)
[4] University of Versailles at Saint-Quentin-en-Yvelines, France

Abstract. Computer scientists are always eager to have a powerful, robust and stable compiler infrastructure. However, until recently, researchers had to either use available and often unstable research compilers, create new ones from scratch, try to hack open-source non-research compilers or use source to source tools. It often requires duplication of a large amount of functionality available in current production compilers while making questionable the practicality of the obtained research results. The Interactive Compilation Interface (ICI) has been introduced to avoid such time-consuming replication and transform popular, production compilers such as GCC into research toolsets by providing an ability to access, modify and extend GCC's internal functionality through a compiler-dependent hook and clear compiler-independent API with external portable plugins without interrupting the natural evolution of a compiler.

In this paper, we describe our recent extensions to GCC and ICI with the preliminary experimental data to support selection and reordering of optimization passes with a dependency grammar, control of individual transformations and their parameters, generic function cloning and program instrumentation. We are synchronizing these developments implemented during Google Summer of Code'09 program with the mainline GCC 4.5 and its native low-level plugin system. These extensions are intended to enable and popularize the use of GCC for realistic research on empirical iterative feedback-directed compilation, statistical collective optimization, run-time adaptation and development of intelligent self-tuning computing systems among other important topics. Such research infrastructure should help researchers prototype and validate their ideas quickly in realistic, production environments while keeping portability of their research plugins across different releases of a compiler. Moreover, it should also allow to move successful ideas back to GCC much faster thus helping to improve, modularize and clean it up. Furthermore, we are porting GCC with ICI extensions for performance/power autotuning for data centers and cloud computing systems with heterogeneous architectures or for continuous whole system optimization.

1 Introduction and Related Work

The compiler is an essential part of modern computing systems responsible for delivering best performing executables across a wide range of architectures quickly and automatically while often satisfying multiple constraints such as code size and compilation time.

Tuning default optimization heuristics of a compiler or optimizing a given program for a given architecture is a tedious, repetitive, error prone, and time-consuming process. In the past few decades, multiple techniques have been developed to improve, automate and speed up this process including empirical iterative feedback-directed compilation [1–11], genetic algorithms and machine learning techniques [12–21], continuous optimization and run-time adaptation [22–28], statistical collective optimization [29, 30] and many other popular methods.

In-house research compilers have been utilized in research for a long time but it is often difficult or even impossible to reproduce their results in realistic environments. Source to source transformation tools such as SUIF [31] and ROSE [32] are also popular to prototype research ideas, however the former is now heavily outdated while the latter is still rapidly evolving, not yet stable enough and is missing some important functionality. We find such frameworks useful for high-level source code manipulation, but we also found that they often have complex interference with the internal optimization heuristics of the coupled source-to-binary compiler making it difficult to analyze final experimental results.

Production proprietary compilers are also regularly used for research. However, they have not been designed to enable prototyping of research ideas, and it is not always easy or possible to access internals of such compilers. Moreover, it is also often impossible to reproduce experimental results in academia without a license. In such cases, researchers may only have access to global compiler flags or some pragmas to tune applications, which may not always be suitable for advanced experimentation.

The LLVM [33] compiler infrastructure also appeared recently targeting both industry and academia, and providing a clear documented API, extension capabilities, JIT, VM, etc. It is gaining popularity but it may still take a long time to provide all available optimizations and support multiple architectures.

GCC [34] is an open-source production compiler that has also been used in research for a while. However, its complexity, often undocumented internals and functions changing from one version to another, long learning curve, rapid evolution, overheads due to frequent synchronization with the mainline compiler and lack of easy extensibility have sometimes prevented it from being used in long-term research projects. Nevertheless, its advantages are very mature and stable multiple front-ends, support for more than 30 families of architectures, GPL license and wide-spread popularity. Moreover, the recently added modular optimization pass manager, experimental polyhedral optimizations (GRAPHITE) and some elementary support for dynamic compilation using CIL [35] and MONO [36] make GCC very attractive for realistic research on code and architecture design and optimizations.

In order to remove some of the above listed disadvantages of production compilers as a research infrastructure and make research developments more portable, we started developing the Interactive Compilation Interface (ICI) [37] to gradually open up compilers and provide access to their internal functionality such as program analysis and optimizations necessary for multiple research projects through a common API and external plugins. It allows quick prototyping of research ideas in a real production environment, potentially saving the effort to build new compiler infrastructure from scratch, while keeping plugin compatibility needed for long-term research projects during natural compiler evolution. GCC maintainers may have some overhead to support such a plugin system, however the GCC community can also benefit from successful research ideas that can be moved back to the compiler immediately. Moreover, it may eventually help to gradually clean up, modularize and document the previously rigid compiler.

ICI has had several major evolutions since 2005 and has been used recently in the long-term MILEPOST project (2006-2009) [19] to add feature extraction passes and enable selection of global optimizations based on popular machine learning techniques. At the beginning it was a compiler-dependent monolithic plugin system, however recently we decided to separate it into 2 parts: low-level compiler dependent plugin system and high-level compiler independent ICI made as a library. The key idea is to update/modify low-level ICI plugin system for different releases of a compiler while keeping high-level ICI reasonably stable to ensure portability of research plugins. In this article, we present further extensions to ICI made during the Google Summer of Code program (GSoC'09) to provide generic function cloning, program instrumentation, pass reordering and control of individual optimizations and their parameters. They are intended to help continue research on various topics including empirical transparent collective optimization [29, 30], run-time program adaptation [25, 21] and code instrumentation, parallelization and scheduling for many-core systems [38, 39, 21].

In the last few years other plugin systems have been proposed and implemented in GCC to enable program analysis, add new passes and control compilation flow [40–42]. Finally, the common agreement has been reached and GCC 4.5 will feature the first common compiler-dependent plugin system. In such case, we can simply substitute low-level compiler-dependent ICI with the native plugin system while keeping high-level ICI compiler-independent that is very important to researchers. We are currently synchronizing our low-level ICI with the plugin system of GCC 4.5 to avoid further duplicate parallel developments. Furthermore, if high-level ICI plugins become stable, they can be easily moved inside the compiler with minimal changes.

The rest of the paper is organized as following: the next section introduces our vision of GCC plugin-enabled research framework, followed by the description of GSoC'09 extensions and some preliminary experimental results. Finally, we briefly describe our attempt to synchronize ICI with the mainline GCC 4.5, followed by a section of conclusions and future work.

2 GCC and collaborative research framework

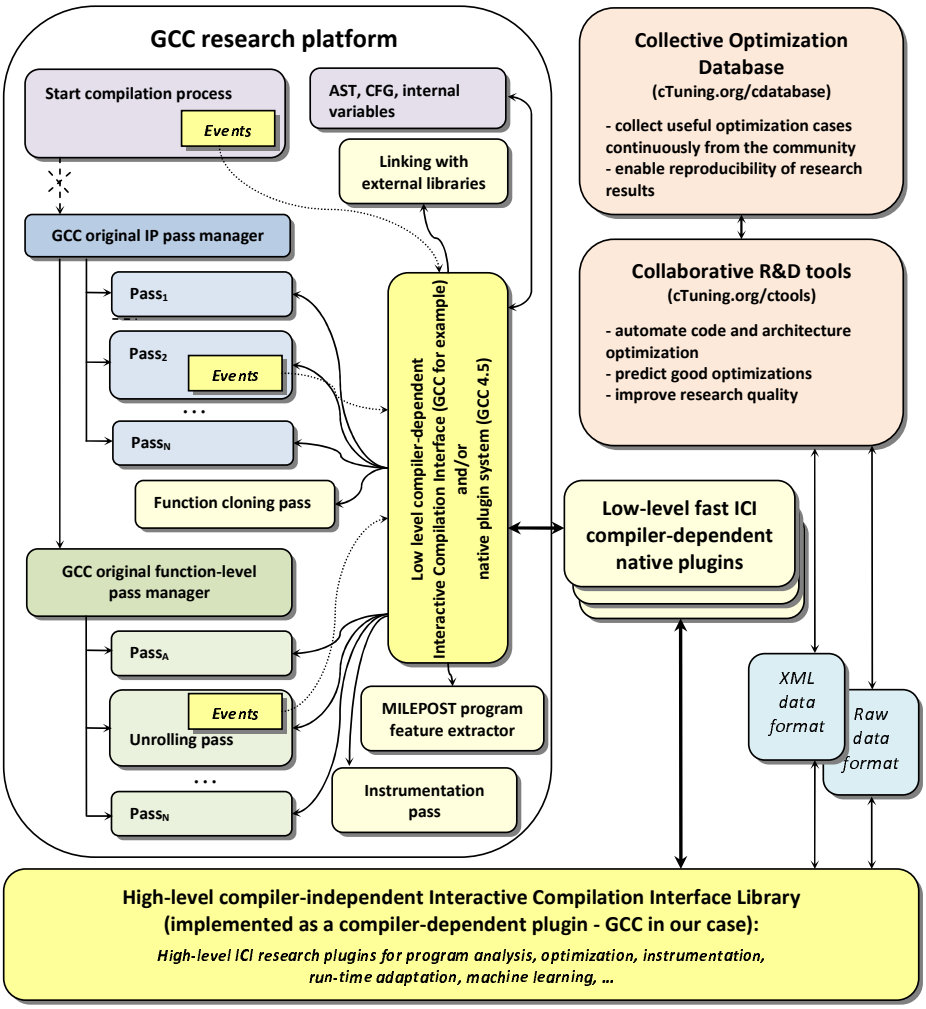


Fig. 1. GCC with high-level compiler-independent and low-level compiler-dependent ICI, and plugins as a research infrastructure connected with cTuning R&D tools and collective optimization database.

We pursue two main goals trying to transform GCC into a research compiler. First of all, we would like to have a common stable extensible compiler infrastructure shared by both academia and industry to improve the quality, practicality and reproducibility of research, and make experimental results immediately useful to the community. Second, we share the long-term vision of

the future adaptive self-tuning computing systems with the cTuning community [43] and therefore continue adding new functionality to GCC with ICI to enable statistical transparent collective optimization [29, 30]. The new ICI functionality provides the ability to substitute the compiler inter-procedural and function-level pass manager with arbitrary sequences of passes. It also includes new passes for generic function-level (and loop-level in the future) cloning, program instrumentation and MILEPOST extractor of static program features [19]. It is now possible to control some transformations such as unrolling individually. Finally, we added support for the XML data format in our GCC plugins to standardize and simplify communication with cTuning tools. Figure 1 shows the extended GCC with low-level compiler-dependent ICI (or native compiler plugin system) and a high-level compiler-independent ICI (that itself is implemented as a low-level plugin/library) connected to the cTuning optimization framework.

3 GSoC'09 extensions to GCC and ICI

This section describes our latest ICI extensions. All sources, plugins, implementation details and experimental results are available at the following collaborative development pages:

- http://ctuning.org/wiki/index.php/CTools:ICI:Projects:GSOC09:Fine_grain_tuning
- http://ctuning.org/wiki/index.php/CTools:ICI:Projects:GSOC09:Function_cloning_and_program_instrumentation

3.1 Enabling full control of GCC passes (selection and reordering)

One of the big problems researchers often face when using GCC is a constantly changing list of passes from one version to another, making their research tools dependent on a specific version of the compiler. We solve this problem by adding new functionality in ICI to obtain a list of available/executing passes and thus make research plugins more portable.

In GCC, all passes are invoked using `execute_one_pass` function. This function tests the pass gate status first and only then executes a pass itself. We added an ICI event call just before this test to send the name of the pass, its parameters and the gate status value to plugins. We wrote a plugin that works in a `record mode` and saves all passes with their original order, parameters and status of the gate.

GCC 4.4.x passes are stored in three linked lists: `all_lowering_passes`, `all_ipa_passes`, and `all_optimization_passes` (the latter is a misnomer). GCC 4.5.x has split the list `all_ipa_passes` into `all_small_ipa_passes` and `all_regular_ipa_passes`, and added `all_lto_gen_passes`. One should note that some of the intra-procedural passes can have function-level sub-passes, so we had to add extra functionality to be able to handle such situations in ICI. Finally, we added ICI event calls before each of these groups and provided a facility to skip execution of those groups of passes in GCC if triggered by the plugin. In

this case, a plugin written in a `reuse mode` can feed all passes back to a compiler in an arbitrary order and also execute auxiliary passes (such as function cloning, program instrumentation or feature extractor) on demand, thus gaining full control of the previously closed and hardwired compilation process.

In the last decade, multiple research projects have been investigating the selection of an optimal order of optimization passes [2, 6] using in-house research compilers. Now, we have a possibility to enhance these studies with a production compiler in a realistic environment but we face a new problem. Since GCC has not been designed for research, it provides very little information about dependencies between passes. This means that we can not explore a large search space of arbitrary orders of GCC passes due to frequent compiler crashes or invalid generated binaries.

(PASS_GROUP=) {*}PASS_GROUP1 { |INDIVIDUAL_PASSA(DEPENDENCE) {,INDIVIDUAL_PASSB(DEPENDENCE){,...}}{&INDIVIDUAL_PASSC(FORBID) {,INDIVIDUAL_PASSD(FORBID)}}}; {*}PASS_GROUP2...; {*}PASS_GROUP3

- PASS_GROUP can be a combination of other pass groups and individual passes;
- "*" means that a group from the right of this sign can be omitted; without it the group will always be selected (for initialization passes, etc).
- "|" means that a group from the left can be selected only if all groups from the right of this sign (separated by ",") have been also selected (true-dependence).
- "&" means that a group from the left can be selected only if none of the all groups from the right of this sign has been previously selected (anti-dependence).
- ";" means that the groups separated by this sign can be selected in any order.

Fig. 2. Formal definition of dependency grammar for GCC passes.

Therefore, we decided to develop a simple dependency grammar to be able to describe and generate valid sequences of optimization passes as described in Figure 2. In the future we plan to represent this grammar in the EBNF (Extended BackusNaur Form) [44], but for simplicity reasons now we use it as is it is presented in Fig. 2. We expect to provide a list of groups of passes with dependencies for each release of a compiler. Now, we have an ability to either generate the default order of passes as in GCC if we turn on all the passes from the list or generate an arbitrary valid sequence of passes for empirical performance/code size/compilation time exploration of various orders. Unfortunately, creating such a list of dependencies based on this grammar is a non-trivial task itself. It is an on-going work and we use both manual and automatic methods to find such dependencies. We start from the default order in GCC and start swapping passes each time checking that the compilation completed successfully and the code produced correct output on a number of datasets thus gradually finding dependencies between passes. We then verify each dependency manually.

Such methodology and grammar can in turn help to modularize GCC and test its correctness (semi-)automatically. We expect to build the first list of passes

with their dependencies for GCC 4.4.x within the next few months. Interestingly, we discovered an explicit dependency between pass "alias", which performs may-alias optimization, and pass "fre", which performs full redundancy elimination: placing "alias" pass after "fre" in some cases could lead to the changes in program semantic and consequently to the errors in produced binary program. In other words, "alias" should be placed always before"fre" lest the compiler could produce invalid code.

3.2 Enabling control of individual transformations

Control over selection of passes and their orders in GCC already opens up many research opportunities. However, our ultimate goal is to provide control over each individual transformation. Previously, special source to source tools have been used to optimize math libraries [1, 4, 8] and large applications [5] using iterative compilation with transformations such as loop tiling, interchange, unrolling and array padding among many others. Most of these transformations are now available inside GCC and other production compilers making them perfect candidates to substitute all specialized tuners.

We patch optimization passes to include event calls just before an individual transformations are applied. We pass all preceding information (decision to apply the transformation based on GCC optimization heuristic including its features and suggested parameters) to a plugin that can either just record this information for further off-line analysis (including machine learning techniques to learn good optimizations) or change the decision and parameters and force the compiler to change its internal decision.

Handling events for each transformation may sometimes slow down the compiler. There can be several solutions to that. We propose including both patched and non-patched optimization passes that can be controlled globally to control individual transformations only on parts of the code where that may have a high payoff in terms of performance or other benefits. We can also create self-adjusting passes that can register/remove events on demand.

We extend ICI to support handling of event parameters. ICI event parameters are actually pointers to temporal data that can live across several events before they are explicitly unregistered. When an event is issued, corresponding handler functions are executed and can read or write event parameters.

With the new ICI, it is fairly easy for researchers to record or reuse parameters of several common data types, such as integer. Pointers can also be registered in ICI as a parameter with the only difference that users have the responsibility to handle the type information correctly.

Together with the control of global optimization passes, the fine-grain control of transformations provides the ultimate control over full compiler optimization heuristic, opening up multiple research opportunities. We currently have support for loop unrolling and loop interchange (from GRAPHITE) and hope to provide support for the rest of transformations together with the community shortly.

3.3 Adding generic function cloning and program instrumentation

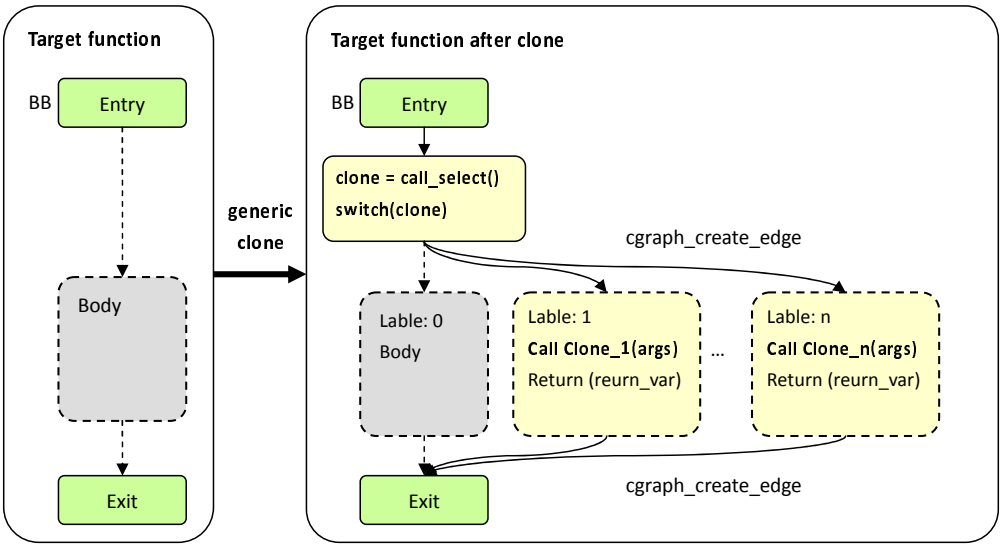


Fig. 3. Generic function cloning pass in GCC.

Multi-versioning helps to make static programs adaptive at run-time. It can be used to enable collective optimization, speed up iterative compilation by evaluating differently optimized versions at run-time and create self-tuning binaries adaptable to different inputs or architectures [25, 30, 29].

Therefore, we have implemented a new pass in GCC, named `generic_clone`, which can generate multiple copies of a given function, and insert a selection function at the beginning of the original function automatically as shown in Figure 3. To enable transparent modification of code (useful for collective optimization), we also add linking with external libraries without Makefile and GCC command-line modifications. These libraries may include different clone selection mechanisms for multiple practical and research purposes. For example, we are porting a clone selection mechanism from [25] to select differently optimized clones using hardware counters to enable adaptation of statically-compiled code for different program and system behavior at run-time. The call to the external selection function is followed by a switch structure to invoke selected clone.

We also have developed an instrumentation pass to be able to modify programs using plugins as shown in Figure 4. Currently, this pass can insert function calls to externally linked libraries at the beginning and the end of the compiled program to support collection of profile information for research tools, collective optimizers and self-tuning programs. We can also add such calls for any function including generated clones. This may be needed to monitor the behavior of the functions using external hardware counter libraries or connect program with

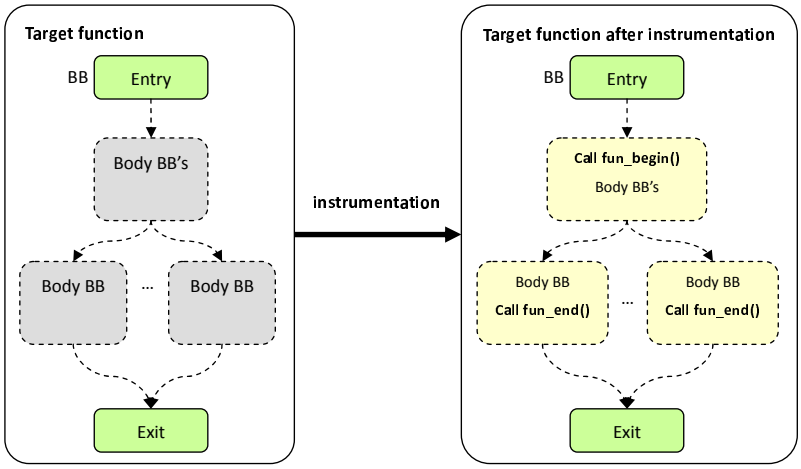


Fig. 4. Program instrumentation pass in GCC.

architecture simulators, etc. Importantly, we can instrument programs through plugins without any modifications to the source code thus keeping programs portable, simplifying development of program analyzers and enabling quick prototyping of research ideas. Eventually, we would like to provide program instrumentation capabilities on loop and even instruction level making GCC a powerful research tool for program analysis.

Both the generic cloning pass and instrumentation pass are implemented as `SIMPLE_IPA_PASS` in GCC and can be executed after the function availability is determined by the `visibility` pass. We added command line options to perform function cloning and instrumentation but we strongly recommend using ICI plugins to invoke these passes while avoiding modifying Makefiles or compilation scripts.

New extensions to GCC make it a powerful toolset to create adaptive binaries and libraries by combing cloning and instrumentation passes with the control of individual transformations to produce clones tuned at fine-grain level through external plugins.

3.4 Adding XML support for plugins data exchange

In the new ICI we decided to add XML support for data exchange between plugins and other tools besides simple row data format. We have several reasons to use XML:

- The format of the data exchanged between plugins and other tools is a well-structured and supports hierarchy.
- The XML format is widely used and can help users utilize their favorite tools to analyze the data.

```
<pass pass_name=''generic_cloning'' pass_type=''SIMPLE_IPA''>
  <function function_filename=''susan.c'' function_name=''susan_thin''>
   <clones >1</clones>
   <clone_name_extension >_clone</clone_name_extension>
   <adaptation_function >clone_select</adaptation_function>
   <options_clone >-O3</options_clone>
  </function>
</pass>
```

Fig. 5. Example of XML data file to perform function cloning using plugins.

```
<pass pass_name=''instrumentation'' pass_type=''SIMPLE IPA''>
  <function function_filename=''susan.c'' function_name=''susan_edges_small''
cloned=''1''>
   <add_function_call_before_func >_instr_start</add_function_call_before_func>
   <add_function_call_after_func >_instr_end</add_function_call_after_func>
  </function>
</pass>
```

Fig. 6. Example of XML data file to perform program instrumentation using plugins.

- The XML format is highly extensible, which is critical for future developments and backward compatibility.
- The XML format can be easily verified for correctness.

Figure 5 shows an example of the configuration file for a function cloning plugin in XML format. In this case, function 'susan_thin' from file 'susan.c' will be cloned once; a clone will be called `susan_thin_clone_1`; a selection function will be inserted calling `clone_select` and -O3 global optimization flag will be applied to a clone.

Figure 6 shows another example for function instrumentation. Function `susan_edges_small` from file 'susan.c' will be instrumented and additional function calls `susan_edges_small_instr_start` and `susan_edges_small_instr_end` will be inserted at the beginning and end of this function.

We are currently synchronizing the XML format for fine-grain program optimizations with the cTuning Collective Optimization Database format [45] to be able to store new experimental data there.

4 Experiments

In order to demonstrate our new research extensions to GCC and show their practicality, we perform several preliminary experiments on program optimization and adaptation (we plan to continue systematic experimentation in future work). For this preliminary study we decided to use both small kernels such as matrix multiply and a few larger applications from the MiBench/cBench benchmark suite [46].

We selected the following popular servers for our experiments:

- Dual-Core AMD Opteron 8218 with Red Hat Enterprise Linux AS release 4 X64_64 (referred later as Opteron machine, cTuning PLATFORM_ID = 11930834698757062);
- Intel Xeon E3110 running CentOS release 5.3, X86_64 (referred later as Xeon machine, cTuning PLATFORM_ID = 395021328416545100, ENVIRONMENT_ID = 7880645273825986);
- Intel Core2Duo T8300 running Linux Ubuntu SMP (referred later as Intel Core2Duo machine, cTuning PLATFORM_ID = 16563583955227076, ENVIRONMENT_ID = 42866903217278407);

Since we are still working on synchronizing our recent developments with mainline GCC, we performed our experiments using GCC 4.4.0 (cTuning COMPILER_ID = 129504539516446542) patched with ICI 2.0 and GSoC'09 extensions. We used the PAPI library [47] to obtain cycle accurate timing of our programs.

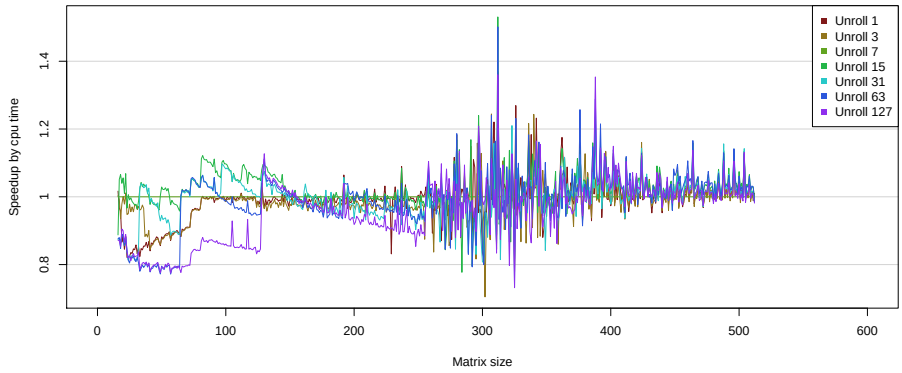
We provide cTuning unique IDs to help reviewers, readers and users verify and reproduce some of our results using cTuning Collective Optimization Database [45]. We hope that the dissemination of experimental results using common R&D tools and optimization repository will become a norm in the future and will help to speed up and improve academic and industrial research.

4.1 Controlling fine-grain program transformations in GCC

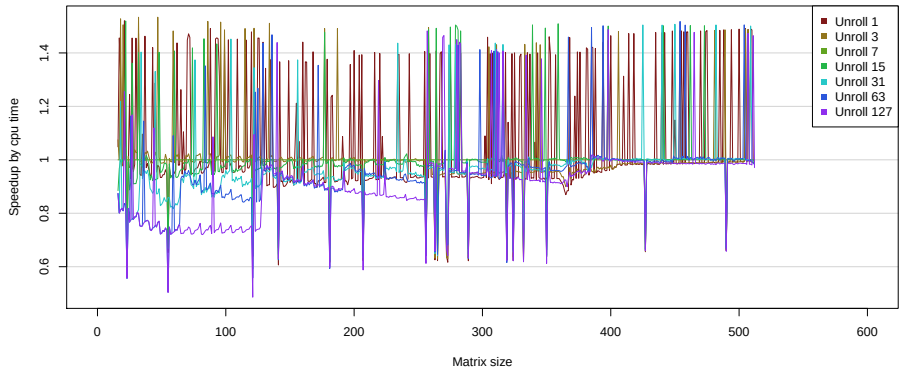
We decided to make a preliminary evaluation of the fine-grained control of transformations in GCC using a very simple and well-known example that up to now often needed specialized source-to-source tuners: optimizing matrix multiply using loop unrolling. Now, we can rely purely on a compiler to create and tune adaptive libraries.

We use the new instrumentation pass to add external cycle accurate timers from PAPI at the beginning and the end of the matrix multiply function. GCC's default unrolling heuristic suggest to unroll matmul 7 times when using `-O3 -funroll-loops` without taking data size into account. Since GCC 4.4 can only make power of two minus one copies of the loop body, i.e., unroll power of two times, we evaluated the following loop unrolling factors: 1, 3, 7, 31, 63 and 127 for square matrix sizes randing from 20x20 to 512x512.

The results from iterative compilation for Opteron and Xeon platforms are presented in Figure 7. They are similar to results obtained through source-to-source transformation from [1, 5, 48]. It clearly shows that the default static compiler optimization heuristic is incapable of producing the best code for a variety of inputs and fine-grain iterative compilation even with only loop unrolling can bring up to 1.5 times speedup. However, it can also bring considerable performance degradation for some combinations of datasets and unrolling numbers. The results for Opteron clearly show correlation of best unrolling factors, with the memory hierarchy showing complex interactions between various cache levels for large matrix sizes. Results for Intel are more difficult to explain and we leave detailed analysis for the future work. However, the results already show that



(a) AMD Opteron



(b) Intel Xeon

Fig. 7. Speedup of matmul for various matrix sizes when controlling loop unrolling in GCC through ICI.

GCC with the new ICI opens up many opportunities for research on fine-grain program optimizations, their interaction (particularly when adding more transformations including polyhedral optimizations) and performance prediction.

4.2 Creating adaptive programs and libraries

The experimental results from the previous section also motivate our static multiversioning approach in GCC to enable creation of adaptive applications. New extensions to ICI allow us to reproduce and extend the research framework from [25, 29, 38, 28] using GCC and select appropriately optimized functions based on the dataset and architecture features (using CPU ID and hardware counters). We will first replicate our technique to build an optimized run-time decision tree automatically using statistical and machine learning techniques as in [28].

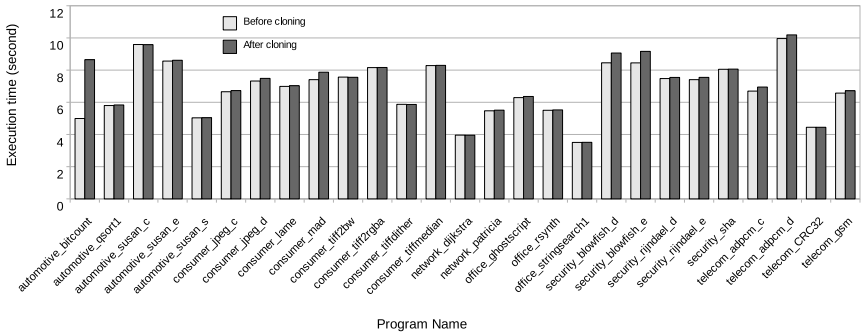


Fig. 8. Overhead of call-switch mechanism during generic function cloning.

Naturally, a run-time overhead may be introduced by our call-switch mechanism. We decided to perform preliminary experiments to evaluate this overhead using cBench benchmark. We selected 6 hot functions covering most of the execution time of all programs using OProfile (excluding `main`), cloned them once and added a cyclical selection mechanism using GCC with new ICI. Figure 8 clearly demonstrates that at least for MiBench/cBench, the execution time overhead of our call-switch mechanism is negligible in most of the cases in comparison with the original code. Figure 9 also shows the negligible growth of binaries after cloning all hot functions, which is critical for embedded systems. These results are similar to results from the [25] when using source-to-source cloning.

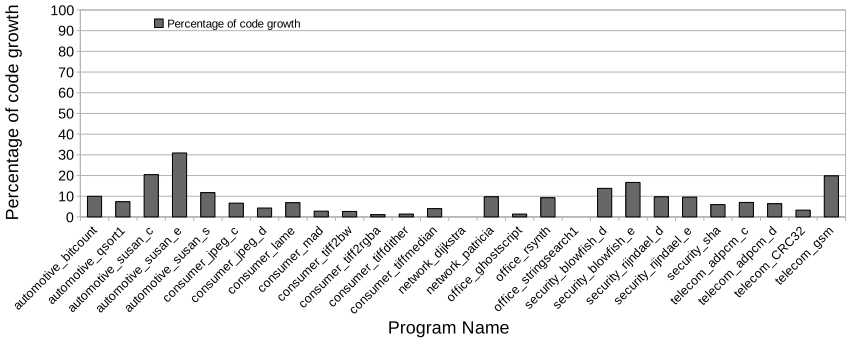


Fig. 9. Binary growth when cloning all hot functions once.

Best flags taken from cTuning database [45] vs baseline(-O3)	susan_e	dijkstra	sha_e
... alias retslot ... crited sink loop ... loopdone vrp ...	28% 762541000430973173	5% 841507490430918931	26% 130797385743093369
... sink **alias** loop ... loopdone vrp ...	34% 127297480343098038	8% 208941853843093041	29% 149436739843093444
... sink **alias** loop ... loopdone **retslot** ... **crited** vrp...	28% 193582669430976651	9% 576051282430931424	31% 350720421430934188

Table 1. Speedups and associated cTuning RUN_ID (to reproduce results if needed) over -O3 for preliminary manual pass reordering experiments using three cBench programs, MILEPOST GCC 4.4.0 with new ICI extensions and Intel Core2Duo machine.

4.3 Preliminary evaluation of pass reordering

Since we have now enabled the optional arbitrary pass selection and ordering in GCC, we would like to evaluate potential performance improvements from different pass sequences. We tried first to reproduce some of the results from [2, 6] but have not succeeded so far. We spent some time manually reordering passes in the "all_optimizations" group and finally found several programs where different pass orders improve the code over the default GCC optimization order and the best selection of optimization passes/flags from cTuning repository [45]. Table 1 shows how the position of pass "alias","retslot" and "crited" influenced overall speedup over -O3, the best default GCC optimization heuristic. Moreover, the most profitable pass sequences depend on the program being optimized. Though this dependence is not yet large, it still shows new research opportunities in GCC and motivates us to extend research from [19] and learn good optimization orders for a given program, architecture and a dataset using statistical and machine learning techniques.

5 Synchronization with mainline GCC 4.5

We have spent more than 3 years on ICI developments and this framework has become too complex to patch for each new release. Since we hope that it can be eventually useful for both research community and GCC end-users, we would naturally like now to move it to mainline GCC. After many discussions, the forthcoming release of GCC will finally feature a low-level plugin framework. However, it is still quite different from ICI. For example, ICI has been written to allow researchers to insert new code easily in random places within GCC without much planning: events and parameters have names and are managed in a hash table which is easy to deal with but may have performance overhead at each event raising site even if there is no callback for that event. In contrast, GCC 4.5 plugins have been designed to have a very low overhead, but require explicitly adding an enum number and a name for every new event, and all parameters have to be passed via a single pointer which may potentially result in many ad-hoc structs. We will try to address this by having a special wrapper to pass a list of named parameters using a va_list* through the original GCC 4.5 events interface. We will also have a number of pre-existing events in GCC 4.5 which we

may want to interface with the ICI named parameters. We also plan to discuss with the GCC community whether the ICI type description could be accepted in GCC. Finally, we separated ICI into high-level compiler-independent research interface and a low-level compiler-dependent fast low-level interface synchronized with the native plugin framework in GCC 4.5.

6 Conclusions and Future Work

In this article we presented our recent GSoC'09 extensions to GCC plugin system to simplify and popularize the use of this free, wide-spread open-source compiler in realistic research on code and architecture optimization. The new infrastructure separates ICI into high-level compiler-independent and low-level compiler-dependent libraries and provides support for generic function cloning and run-time adaptation for statically-compiled programs in heterogeneous environments, inter-procedural and function-level optimization pass selection and reordering with a dependency grammar able to describe valid sequences, control of individual transformations and their parameters for fine-grain application optimization, and the XML representation of the compilation flow to ease communication with external tools.

We are currently synchronizing the low-level Interactive Compilation Interface and GSoC'09 extensions with mainline GCC and its new native plugin framework to provide a reasonably stable compiler-independent API to the research community during rapid compiler evolution. We will be gradually adding external control of OpenMP and individual transformations including inlining, vectorization and polyhedral loop transformations from the GRAPHITE pass. We plan to provide support for program instrumentation and instruction manipulation for advanced code analysis and optimization. Eventually, researchers would also like to have source to source transformations in GCC as well as support for dynamic optimization and split compilation (using MONO and GCC4CIL, for example), remove hard-coded dependencies between passes, and exploit direct access to global variables. Finally, we would like to start systematic investigation of the correctness of automatically generated combinations of optimizations. This is of particular importance during statistical collective optimization [29] when using the cTuning framework with GCC [30, 43] for embedded devices, data centers and cloud computing systems for automatic, continuous and transparent performance/power tuning of user applications or for whole system optimization (such as Moblin and Android).

7 Acknowledgments

Yuanjie Huang and Liang Peng have been supported by Google Summer of Code program'09 program to implement fine-grain tuning, function cloning and program instrumentation. Yuriy Kashnikov has been supported by UVSQ to implement pass reordering in GCC. Joern Renneke has been supported by INRIA to move the Interactive Compilation Interface to mainline GCC and synchronize it with the current GCC 4.5

plugin system. We would like to thank multiple users from GCC, cTuning and HiPEAC communities for their useful feedback. We would also like to thank Prof. William Jalby for interesting discussions about ICI and program optimizations. Finally, we would like to thank anonymous reviewers, Jeremmy Bennett and Phil Barnard for their insightful comments to improve this article.

References

1. Whaley, R., Dongarra, J.: Automatically tuned linear algebra software. In: Proceedings of the Conference on High Performance Networking and Computing. (1998)
2. Cooper, K., Schielke, P., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES). (1999) 1–9
3. Bodin, F., Kisuki, T., Knijnenburg, P., O'Boyle, M., Rohou, E.: Iterative compilation in a non-linear optimisation space. In: Proceedings of the Workshop on Profile and Feedback Directed Compilation. (1998)
4. Matteo, F., Johnson, S.: FFTW: An adaptive software architecture for the FFT. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing. Volume 3., Seattle, WA (May 1998) 1381–1384
5. Fursin, G., O'Boyle, M., Knijnenburg, P.: Evaluating iterative compilation. In: Proceedings of the Workshop on Languages and Compilers for Parallel Computers (LCPC). (2002) 305–315
6. Kulkarni, P., Zhao, W., Moon, H., Cho, K., Whalley, D., Davidson, J., Bailey, M., Paek, Y., Gallivan, K.: Finding effective optimization phase sequences. In: Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES). (2003) 12–23
7. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.: Compiler optimization-space exploration. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). (2003) 204–215
8. Singer, B., Veloso, M.: Learning to predict performance from formula modeling and training data. In: Proceedings of the Conference on Machine Learning. (2000)
9. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). (2006) 319–332
10. Heydemann, K., Bodin, F.: Iterative compilation for two antagonistic criteria: Application to code size and performance. In: Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems, colocated with CGO. (2006)
11. Hoste, K., Eeckhout, L.: Cole: Compiler optimization level exploration. In: Proceedings of International Symposium on Code Generation and Optimization (CGO). (2008)
12. Nisbet, A.: GAPS: Genetic algorithm optimised parallelization. In: Proceedings of the Workshop on Profile and Feedback Directed Compilation in conjunction with PACT'98. (1998)
13. : ACOVEA: Using Natural Selection to Investigate Software Complexities. `http://www.coyotegulch.com/products/acovea`
14. : Learning to schedule straight-line code. In: Proceedings of the Conference on Neural Information Processing Systems (NIPS). (1997)

15. Monsifrot, A., Bodin, F., Quiniou, R.: A machine learning approach to automatic production of compiler heuristics. In: Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications. LNCS 2443 (2002) 41–50
16. Stephenson, M., Martin, M., O'Reilly, U.: Meta optimization: Improving compiler heuristics with machine learning. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). (2003) 77–90
17. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO). (2005)
18. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M., Thomson, J., Toussaint, M., Williams, C.: Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). (2006)
19. Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., O'Boyle, M.: Milepost gcc: machine learning based research compiler. In: Proceedings of the GCC Developers' Summit. (June 2008)
20. Dubach, C., Jones, T.M., Bonilla, E.V., Fursin, G., O'Boyle, M.F.: Portable compiler optimization across embedded programs and microarchitectures using machine learning. In: Proceedings of the 42nd International Symposium on Microarchitecture (MICRO). (December 2009)
21. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI). (2009)
22. Voss, M., Eigenmann, R.: Adapt: Automated de-coupled adaptive program transformation. In: Proceedings of the International Conference on Parallel Processing (ICPP). (2000)
23. Lu, J., Chen, H., Yew, P.C., Hsu, W.C.: Design and implementation of a lightweight dynamic optimization system. In: Journal of Instruction-Level Parallelism. Volume 6. (2004)
24. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO), Palo Alto, California (March 2004)
25. Fursin, G., Cohen, A., O'Boyle, M., Temam, O.: A practical method for quickly evaluating program optimizations. In: Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC). Number 3793 in LNCS, Springer Verlag (November 2005) 29–46
26. Stephenson, M.W.: Automating the Construction of Compiler Heuristics Using Machine Learning. PhD thesis, MIT, USA (2006)
27. Lau, J., Arnold, M., Hind, M., Calder, B.: Online performance auditing: Using hot optimizations without getting burned. In: Proceedings of the ACM SIGPLAN Conference on Programming Languaged Design and Implementation (PLDI). (2006)
28. Luo, L., Chen, Y., Wu, C., Long, S., Fursin, G.: Finding representative sets of optimizations for adaptive multiversioning applications. In: 3rd Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'09), colocated with HiPEAC'09 conference. (January 2009)

29. Fursin, G., Temam, O.: Collective optimization. In: Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009). (January 2009)
30. Fursin, G.: Collective tuning initiative: automating and accelerating development and optimization of computing systems. In: Proceedings of the GCC Developers' Summit. (June 2009)
31. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.A.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: Suif: An infrastructure for research on parallelizing and optimizing compilers. SIGPLAN Notices **29**(12) (1994) 31–37
32. : ROSE Compiler Infrastructure. `http://www.rosecompiler.org`
33. : LLVM Compiler Infrastructure. `http://llvm.org`
34. : GNU Compiler Collection. `http://gcc.gnu.org`
35. Cornero, M., Costa, R., Pascual, R.F., Ornstein, A.C., Rohou, E.: An experimental environment validating the suitability of cli as an effective deployment format for embedded systems. In: Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC). (January 2008)
36. : MONO: cross platform, open source .NET development framework. `http://www.mono-project.com`
37. : ICI: Interactive Compilation Interface: plugin system to convert production compilers into research toolsets (2005)
38. Jimenez, V., Gelado, I., Vilanova, L., Gil, M., Fursin, G., Navarro, N.: Predictive runtime code scheduling for heterogeneous architectures. In: Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009). (January 2009)
39. Long, S., Fursin, G., Franke, B.: A cost-aware parallel workload allocation approach based on machine learning techniques. In: Proceedings of the IFIP International Conference on Network and Parallel Computing (NPC 2007). Number 4672 in LNCS, Springer Verlag (September 2007) 506–515
40. Starynkevitch, B.: Multi-stage construction of a global static analyser. In: GCC Developers' Summit. (July 2007)
41. Glek, T., Mandelin, D.: Using gcc instead of grep and sed. In: Proceedings of the GCC Developers' Summit. (June 2008)
42. Sean Callanan, D.D., Zadok, E.: Extending gcc with modular gimple optimizations. In: GCC Developers' Summit. (July 2007)
43. : cTuning.org: Collective tuning center to automate design and optimization of computing systems. `http://cTuning.org` (2008)
44. Whitney, G.: An extended bnf for specifying the syntax of declarations. In: AFIPS '69 (Spring): Proceedings of the May 14-16, 1969, spring joint computer conference, New York, NY, USA, ACM (1969) 801–812
45. : cTuning optimization repository (Collective Optimization Database). `http://ctuning.org/cdatabase`
46. : Collective Benchmark: collection of open-source programs and multiple datasets from the community. `http://ctuning.org/cbench`
47. : PAPI: A Portable Interface to Hardware Performance Counters. `http://icl.cs.utk.edu/papi`
48. Fursin, G.: Iterative Compilation and Performance Prediction for Numerical Applications. PhD thesis, University of Edinburgh, United Kingdom (2004)